Software Verification - Part 1

M2 informatique - parcours VL

Université de Bordeaux

Vincent Penelle

2024-2025

Contents

Chapter 1

Introduction - A quick overview of verification and the course

1.1 Motivation

Bugs are undesirable in a program, especially in critical systems whose failure would be costly (in lives, resources, or something else). In such a case, it is desirable to spot bugs before the program is executed in real context.

A widespread approach is to test a program on input which are deemed representative of the behaviour of the program. This approach is called *validation*. It relies on devising enough scenarios and observe that the program behaves as expected on these scenarios. If not, a bug is spotted and can be corrected. However, such an approach can only show the presence of bugs, not their absence, as the method relies on the fact that the scenarios are representative of the behaviour of the program, and thus depends on the (usually human) input of these scenarios. While sufficient for a good confidence in most programs, for critical software, such an approach might not be enough, and a lot of well-known bugs escaped such detection (e.g., Ariane 5 (litteral) crash, or more recently, the SSL bug hartbleed).

A symmetrical approach, called *verification* has been concurrently developped, whose goal is to prove the absence of bugs. The technique we will use in this course is called *static analysis*, which relies on analysing the code before compiling it. Il also have the goal of being *sound*, meaning never deeming a program correct when it is not. That means that if our tool says a program is correct, it is surely correct, but it might say a correct program

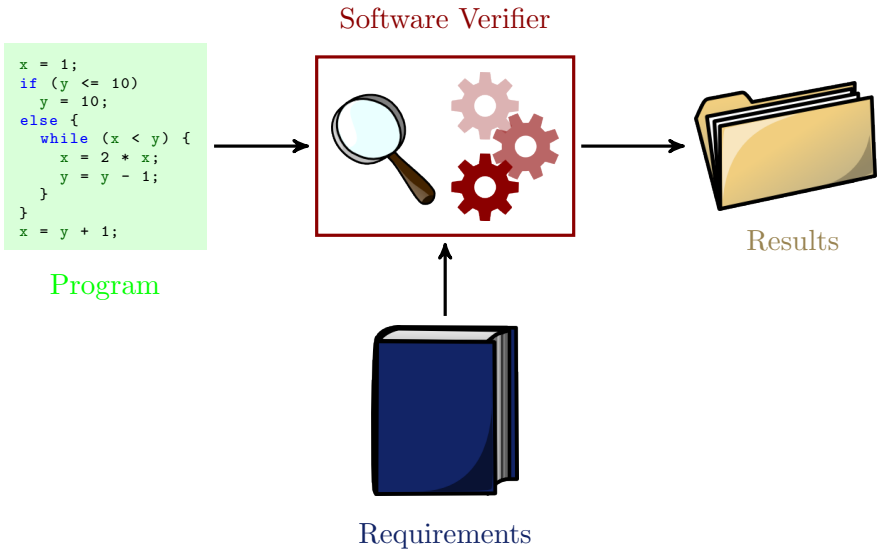

Figure 1.1: An ideal (and impossible) software verifier

is incorrect.

1.2 Ideal Goal and limits

The problem we want to answer is that a given program, given by its code, satisfies a *specification*, given in a logic (which we'll tell more about later). This problem is called the *model-checking* problem. Formally, it can be presented this way:

Input: A program P and a specification φ.

Output: Do all executions of P satisfy φ?

Of course, the exact formulation of the question will depend on the logic in which φ is written, and we will need to introduce more formalism to express formally what it means to be satisfied by an execution.

What we would like ideally is to have a program answering this problem, as depicted in Figure 1.1.

However, such a goal is impossible without restricting what we aim for. Indeed, it is well-known that the halting problem of a Turing machine is undecidable, and as presented earlier, the model-checking problem can express

that a program represtented by a Turing machine halts. A formulation of that fact is Rice Theorem:

Theorem 1.2.1 (Rice's Theorem)**.** *Any non-trivial semantic property of programs is undecidable.*

Another idea would be to notice that the number of possible execution of a program on an actual machine is finite, and it is sufficient to check all these execution. However, the number of such execution is so astronomically high (around $10^{2400000}$ states for 1 MB of memory) that in practice it is impossible to do so (if it were, that would be systematicizing validation).

We therefore need to do compromises on our goal to get usable tools.

1.3 Compromises

It is not because it is impossible to get a verifier that answers to every model-checking instances that we cannot get verifiers that answers to some model-checking instances. We will thus conceive verifiers which are *sound* in the sense that if the verifier answers a programs satisfies the specification, it is proven that it does, but might not guarantee that a negative answer proves the presence of a bug (which would be called *complete*). To get such a verifier several approaches are possible, we describe a few here. Of course, these approaches are not mutually exclusive (on the contrary), so we might use several of them in some algorithm.

1.3.1 Incomplete algorithms

This approach consists in having a program that instead of applying a decision algorithm, applying a *semi-decision* algorithm and stopping it after a bound on the time (to not get stuck on an infinite loop). In this approach, we might get three answers to the model-checking problem: *yes*, *no* and *unknown*. Of course, we will try to get algorithms in which *unknown* occurs as little as possible. In our course, the bounded model-checking method will be an exemple of such a method.

1.3.2 Restricting the semantic

On Turing-complete programs, the Theorem of Rice states that we cannot decide any property. But on model with less expressivity, this theorem does not hold, and some properties are decidable. This approach consists in abstracting the semantic of the program in a less expressive model and

decide the model-checking on that abstraction. Of course, to be coherent, the abstraction will need to be *sound* in the sense that it will need to ensure that if the model-checking returns yes on the abstraction, it would have returned yes on the concrete program. Therefore the abstraction might depend on the specification (or rather the class of specification).

1.3.3 Restricting the specification

This technique will always go alongside the others because of Rice Theorem. It consists in restricting the type of specification we will accept as input of the verifier. Typically, that consists either in restricting the logic in which the specification is expressed (e.g., considering First-Order Logic instead of Monadic Second-Order Logic) or only consider questions like reachability or liveness. In this course, we will focus on the reachability question, which asks whether a given position of the program is reachable or not, the target position representing a bug.

1.3.4 Summary and complexity considerations

In summary, the problem we will focus on in this course, can be reformulated as the following: given an abstraction function $\mathcal{A}$,

Input: A program P and a position q of P.

Output: Whether $\mathcal{A}(q)$ is reachable in $\mathcal{A}(P)$.

Furthermore, we might allow the algorithm answering this problem to only be a semi-algorithm.

Of course, this might seem as an unacceptable compromise with truth, as we will accept that we do not always answer faithfully the problem. However we cannot expect more and it is better to be able to answer sometimes than never. Furthermore, our soundness obligation ensures that we will never promise a program is correct when it isn't, which will allow to effectively validate some programs, which is what we need for safety of critical system. This is actually the converse than testing, which can never guarantee absolutely that a program is correct. Actually, it is usually desirable to use both techniques alongside.

Finally, there is a last thing to consider: complexity. Usually, the more a technique is close to the correct answer, the worse its complexity will be. And complexity of model-checking (or reachability) algorithm is usually not that good, except on very restricted model (exponential complexity or worse

is far from uncommon). It will therefore be sometimes very efficient to have coarse algorithm which are not very precise (while being sound) but have good complexity. They will allow to quickly check some part of the program and deem them correct, and may even lead to the direction to explore to either find a bug or show the program is correct with an other analysis. The third part of the course will actually concern such techniques.

1.4 Overview of this course

1.4.1 Summary

The course is separated in three parts. The goal is to present several verification techniques through the implementation of a tool, Simple Program Analyser.

Introduction and Bounded Model-Checking

The first part will first introduce broadly verification (this chapter). Then it presents the tool we use in background of ours to automatically decide properties (SMT-solvers), the language of programs we will verify and their formal semantics. Finally, it presents the technique of Bounded Model-Checking, which is an imcomplete method.

It will be taught by Vincent Penelle, and is summarised in the present document.

Abstract Interpretation

The second part focuses on abstract interpretations, i.e., a way of astracting the semantics of the program in an abstract domain in which the reachability set of the program is effectively computable.

It will be taught by Grégoire Sutre.

CEGAR and counter-systems

The third part will present an approach called Counter-Example Guided Abstraction Refinement, which starts by studying a very coarse abstraction of the program, and if that analysis finds a false counter-example, use that false counter-example to find a abstraction that excludes it and start over. It will also present some results on the reachability over counter-systems and vector addition systems.

It will be taught by Jérôme Leroux.

1.4.2 Summary of the first part

The rest of the document is separated in three parts:

- A definition of the SAT Modulo Theory problem and an overview of SMT-solvers.
- A definition of the programming language we are using and its semantics.
- The definition of the Bounded Model-Checking problem and two algorithm to solve it.

Alongside this part are one exercice session on modelling with SAT and SMT, and four practice sessions, one on implementing a solver to familiarize with SMT, one on implementing the semantics of our programming language, and two on implementing the two algorithms for Bounded Model-Checking.

Chapter 2

Tool for deciding: SMT-solvers

The goal of this chapter is to present the problem of SAT Modulo Theory, and explain how it can be used for verification purposes. To do that, we first recall SAT problem, as it is closely related.

2.1 SAT

2.1.1 Definition

Propositional Logic

Propositional Logic is a logic which deals with Booleans.

We first define the Boolean set as $\mathbb{B} = \{\bot, \top\}$, and three connectors on it $\wedge$, $\vee$ and $\neg$ whose semantics is recalled in Figure 2.1.

We also fix a (possibly infinite) set of *variables* Var, and we define a *propositional formula* with the following grammar:

$$\varphi ::= x \in \text{Var} \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi'$$

$\wedge$	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$

$\vee$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$

	$\neg$
$\bot$	$\top$
$\top$	$\bot$

Figure 2.1: Boolean connectors

For the sake of simplicity, we do not include more in the definition, but for clarity, we will allow the use of usual connectors and constant in formulae ($\Rightarrow$, $\Leftrightarrow$, $\top$, $\bot$, etc).

An *assignment of variables* or *valuation* is a function $\nu : \text{Var} \to \mathbb{B}$. We define the evaluation of a formula φ in a valuation ν recursively:

- $[\![x]\!]_\nu = \nu(x)$.
- $[\![\varphi \wedge \varphi']\!]_\nu = [\![\varphi]\!]_\nu \wedge [\![\varphi']\!]_\nu$.
- $[\![\varphi \vee \varphi']\!]_\nu = [\![\varphi]\!]_\nu \vee [\![\varphi']\!]_\nu$.
- $[\![\neg\varphi]\!]_\nu = \neg[\![\varphi]\!]_\nu$.

If $[\![\varphi]\!]_\nu = \top$, we write $\nu \models \varphi$.
The SAT problem is the following:

Input: A propositional formula φ.

Output: Is there an assignment of variable such that $\nu \models \varphi$?

2.1.2 SAT and verification

It is a well-known fact that SAT is an NP-complete problem. The interesting feature of SAT is that despite its NP-completeness, there are a lot of efficient algorithms that solves it. Therefore, an interesting strategy to solve a problem is to *encode* it in a propositional formula through a *reduction*, solve the SAT problem with a SAT-solver, and in case of satisfiability, use the assignment constructed to get an information on the input problem.

As you know if you have a polynomial reduction to SAT from a problem P, you prove that P is in NP, but you can actually still use a non-polynomial reduction to SAT as an algorithm to decide a problem (it just won't be an NP algorithm).

A strategy could be to encode our reachability problems in SAT and use a SAT solver. However, when we deal with infinite domains, as program do, it won't be possible to do so. SAT modulo theory will be an answer to that.

But before moving to it, let us first quickly detail the core algorithm of SAT solvers, as it is an algorithm useful for verification in itself.

2.1.3 DPLL algorithm

This algorithm was introduced by Martin Davis, Hilary Putnam, George Logemann and Donald Loveland (hence its name). It is basically a deep

Algorithm 1: DPLL(φ, ν)

Input: A CNF formula $\varphi = C_1 \wedge \cdots \wedge C_k$, and a variable assignment $\nu : \text{Var} \to \mathbb{B}$

Output: "Not satisfiable" or "Satisfiable with $\nu' : \text{Var} \to \mathbb{B}$"

```
(φ', ν') ← (φ, ν)
repeat
    (φ, ν) ← (φ', ν')
    (φ', ν') ← UnitPropagation(φ, ν)
    (φ', ν') ← PureLiteralElimination(φ', ν')
until (φ', ν') == (φ, ν)
if φ == ⊤ then
    return "Satisfiable with ν"
if φ == ∅ then
    return "Not Satisfiable"
(ℓ, b) ← GuessNext(φ, ν)
if DPLL(φ, ν[ℓ ← b]) == "Satisfiable with ν''" then
    return "Satisfiable with ν''"
return DPLL(φ, ν[ℓ ← ¬b])
```

first search algorithm (the branching being the guessing of truth values of variable), but with two very important twists : first, before guessing, it simplifies the formula with *unit propagation* and *elimination of pure literals*, and secondly, it relies on heuristic to guess which variable to which attribute a value (and which one). We will detail here the first point, but not the second (the heuristic involved are different from one solver to another and complicated to explain here, but are at the center of the efficiency of SAT solvers). Of course, in the worst case, this algorithm has exponential complexity (I hope you had guessed it), though on most formulæ, SAT solvers are insanely efficient. Its pseudo-code is given in Algorithm 1.

Unit Propagation and Elimination of Pure Literals

The unit propagation simply observes that if a clause is reduced to a single literal, the only possibility for the formula to be satisfied is that this literal is true. Therefore, if a formula is of the form $\ell \wedge C_1 \wedge \cdots \wedge C_k$, we Unit propagation puts ℓ to $\top$ in ν, and replaces ℓ with $\top$ in $C_1, \cdots, C_k$ and simplifies them. Otherwise Unit Propagation does nothing.

The elimination of pure quantifiers makes another simple observation :

if a variable only appears positively (resp. negatively), we can simply put it to true (resp. false), and all clauses containing it will be satisfied. Therefore, if a formula is of the form $(\ell \vee C_1) \wedge \cdots \wedge (\ell \vee C_k) \wedge C'_1 \wedge \cdots \wedge C'_{k'}$ (where $\neg\ell$ does not appear), Elimination of Pure Quantifiers puts ℓ to $\top$ in ν, and considers the formula $C'_1 \wedge \cdots \wedge C'_{k'}$. Otherwise it does nothing.

2.2 SAT Modulo Theory (SMT)

As propositional logic cannot handle infinite domain, we will use first-order logic to handle them. However, the full first-order logic being indecidable in general, we will turn to subsets of it which enjoy decidability.

2.2.1 Definition

For the rest of the section, we fix an universe $\mathcal{U}$ which will (informally) represent the domain of the program. We do not ask that this universe has any property, and we will actually consider that it can be heterogeneous. For exemple, if a program manipulates both integers and floats, we will consider our universe contains both of them. We will discuss what this implies later. For now, for the sake of simplicity, we just define FO over an arbitrary universe $\mathcal{U}$.

As before, we fix a (possibly infinite) set of variables Var.

A *predicate* of arity n is function from $\mathcal{U}^n$ to $\mathbb{B}$. We denote $\mathcal{P}$ the set of predicates over $\mathcal{U}$, and $\mathcal{P}_n$ the set of predicates of arity n.

A good example of predicates (and the one we will focus in this course) are arithmitic inequalities. For example, $x + 3 == y$ is a predicate of arity 2, as it has two variables.

We can now define first-order logic over $\mathcal{U}, \mathcal{P}$ (FO$[\mathcal{U}, \mathcal{P}]$), through the following grammar:

$$\varphi ::= P(x_1, \cdots, x_{|P|}) \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \exists x, \varphi \mid \forall x, \varphi$$

where $P \in \mathcal{P}$, $|P|$ is its arity, and $x_1, \cdots x_{|P|} \in$ Var. The red part represent the quantifiers. Removing it from the grammar gives the quantifier-free fragment of FO.

We define *valuations* similarly as before as functions $\nu :$ Var $\to \mathcal{U}$. Given a valuation ν, a variable $x \in$ Var and an integer z, we define the substition of x by z in ν as the valuation $\nu[x \leftarrow z]$ such that, $\nu[x \leftarrow z](x) = z$, and for all variable y different from x, $\nu[x \leftarrow z](y) = \nu(y)$. The evaluation of a formula φ in a valuation ν is defined recursively:

- $[\![P(x_1, \cdots, x_{|P|})]\!]_\nu = P(\nu(x_1), \cdots, \nu(x_{|P|}))$.
- $[\![\varphi \wedge \varphi']\!]_\nu = [\![\varphi]\!]_\nu \wedge [\![\varphi']\!]_\nu$.
- $[\![\varphi \vee \varphi']\!]_\nu = [\![\varphi]\!]_\nu \vee [\![\varphi']\!]_\nu$.
- $[\![\neg\varphi]\!]_\nu = \neg[\![\varphi]\!]_\nu$.
- $[\![\exists x, \varphi]\!]_\nu = \top$ if and only if there exists $u \in \mathcal{U}$, such that $[\![\varphi]\!]_{\nu[x \leftarrow u]} = \top$.
- $[\![\forall x, \varphi]\!]_\nu = \top$ if and only if for all $u \in \mathcal{U}$, $[\![\varphi]\!]_{\nu[x \leftarrow u]} = \top$.

If $[\![\varphi]\!]_\nu = \top$, we write $\nu \models \varphi$.
We consider the problem of satisfiability of such formulae:

Input: A formula φ of FO$[\mathcal{U}, \mathcal{P}]$

Output: Does there exists a valuation ν such that $\nu \models \varphi$?

Contrary to the propositional case, this problem is *undecidable* in an arbitrary universe, even in the quantifier-free fragment. Which would seem to seal the case of its use in verification. Yet we're not done yet.

2.2.2 SMT-solvers

Though the satisfiability problem for FO is undecidable in general, we can still do something.

First, we can always try to use only sets of predicates on which the problem is decidable. For example, it is the case of Presburger arithmetic, in which we manipulate integers, and only allow for addition and substraction between variables (i.e., $x + y = z - 3$ is a predicate of Presburger arithmetic, as well as $2x = 6 - y$ (as multiplying by a constant is an addition), but $xy = z$ is not).

An other option is to use incomplete methods, as described in the introduction section, i.e., use semi-algorithms which we allow to run up to a timeout and allow the tool to answer unknown if the timeout is reached. When dealing with undecidable logic, that is better than nothing.

In these two cases, there is a need for efficient algorithms to solve the satisfiability problem for FO, that so-called SMT-solver strive to implement. As the SAT problem is well-understood and has efficient algorithms, most techniques for SMT borrow idea from it, or even use SAT solvers as backend, limiting the solving of propositions in the theory to atoms.

A first idea, called *eager* consist in trying to encode a FO-formula into full SAT formula (which depending on the theory will obviously be not polynomial in size, if at all possible).

Another technique that is used in most SMT-solvers, called *lazy*, consists in "cutting" the resolution of the problem in two distinct parts:

- The propositional part, i.e., the formula where the predicates are replaced with variables, and use the efficients SAT-solving methods to determine possible truth values of the atoms.
- The theory parts, in which, for each predicate, depending on the universe and the set of predicates (called theory) it belongs to, runs a decision or semi-decision algorithm to determine if it is satisfiable or not (or unknown). The theory solver is dependent on the theory, many different algorithm exist, and their efficiency varies greatly, depending on the theory (for example, if you can encode an arithmetic problem without variable multiplication, solvers will usually behave better). Thanks to some techniques, it is also possible to use solvers from different theories on a same formula.

The solver thus strives to make the two parts communicate by using the theory part only on assignations that would satisfy the propositional part, and thus ease the computational cost (the theory solver is usually costly). The broad idea is to first put the formula in CNF, separating the different theories properly, and then run a modified version of DPLL that uses unit propagation (but not pure literal elimination, as the literals might not be independent), and when finding a possible assignment of variable that satisfies the propositional part, run the theory solver, and if an inconsistency is returned, uses it to enrich the formula with a new clause (to avoid rechecking the same bad properties many times), backtracks and continue the algorithm.

Of course, the tools are more complicating than what is summarised here, the interaction can occur more early than on a full assignment for example, and the theory can be used to cut whole branches that would otherwise be explored (basically, by implementing an alternate version of DPLL), but the base idea is still the one sketched above. The process is also complexified in the presence of quantifiers, but most SMT solvers have a way to deal with them.

A last advantage of this lazy method is that it is possible to extract a satisfying assignment of a satisfied formula, and most SMT-solver are actually able to do that. (to complete)

There are several SMT-solvers that exists and have different strength and weakness. In this course, we will use Z3, developped by Microsoft Research, namely through its Ocaml interface. Though it's not the only existing, among others we could cite Alt-Ergo (developped by OcamlPro, a french company), or Barcelogic (from the UPC of Barcelona). To facilitate comparison, and ultimately improvement of SMT-Solvers, they all implement the so-called SMT-Lib which allows to provide normalised inputs to the solvers. Thus, they can all compare on the same benchmarks, and can contribute to develop new libraries for Theories which are not yet well supported.

The main structures supported by SMT-Libs are Uninterpreted Functions (only equality), Boolean, Integers, Reals, Bitvectors of fixed size, Arrays and Inductive Data Types. On these structures, cohabit several theories, depending on the form of the predicates used in the formula (as noted before, an arithmetic formula over integer which doesn't include multiplication between variable can be treated in Presburger arithmetic, which, contrary to full arithmetic, is decidable).

More details about SMT solvers can be found in [1], and on slides by one of the authors at a summer school[1]

2.2.3 SMT and Verification

So, why did we discuss SAT and SMT?

The reason is that, despite its high complexity, or even undecidability depending on the theory, SMT-solver provide efficient algorithms, and the satisfiability problem is actually very similar to the reachability problem we want to study. Therefore, rather than implementing an ad-hoc algorithm that would not necessarily be as efficient, a strategy we will use for deciding the reachability problem is to encode it as an FO formula, and call Z3 on it.

Our strategy will be the following:

- Provide an automata-like presentation of the program (Chapter 3)
- Encode the successor relation of that automaton as a FO-formula.
- Translate the question of reachability into a formula (using the successor relation), and asks the answer to a SMT-solver.

Of course, that is a broad picture of what we will do, and depending on the part of the course, the encoding of the successor relation will not

[1] `https://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa08/slides/barret2_smt.pdf`

be the same. More precisely, some parts of the decision procedure can be kept out of the formula but treated with an ad-hoc algorithm (that will be the case with the depth-first search algorithm of the bounded model-checking). But whatever our strategy is, the successor relation will give arithmitic constraints, and that is mainly the resolution of those we will ask to a solver.

2.3 Implementation

The exercise session linked to this chapter consist in familiarising with the Ocaml API of Z3 through the implementation of a logic game necessitating to handle first-order logic.

Chapter 3

A minimal Programming Language

In this chapter, we present the language we will work on. Basically, the language we use is C-like, but ripped of features that, while being important for a programmer, would complicate our work without changing the principles we are dealing with. The goal of this course being to present the verification technique and show that they are applicable, we restrict ourselves to a simpler language in order to have the time to implement a simple tool during the course. Note two points, though. First, despite being simpler in syntax, this language remains Turing-complete, and therefore still serves as a proof of concept that verification is implementable. Secondly, to develop a tool usable for actual program, we would need to include the language feature we let aside for now. But some tool dealing with the full language exists, like frama-c for example, though it focuses on Weakest Precondition rather than reachability property.

3.1 A restricted language

Our program will only manipulate integers. Though SMT-solver can handle reals, floats are actually not reals and pose a variety of difficulties for verification purposes and handling them would require a full course on that problem. We thus let them aside, and consider our programs manipulate only integers. Similarly, we let aside pointers, as the handling of dynamic memory is also source of many problems. Here, we will only consider programs as a single unit (no function), and therefore all memory will be modelled as variables.

We thus first define Arith, the set of arithmetic expressions over integers and variables (in a set Var), which will be the core of our programming language.

$$t \quad ::= \quad x \in \text{Var} \;\mid\; z \in \mathbb{Z} \;\mid\; t+t \;\mid\; t-t \;\mid\; t \times t \;\mid\; t/t \;\mid\; t\%t$$

Given an arithmetic expression t and a variable assignment $\nu : \text{Var} \to \mathbb{Z}$, we recursively define the value of t in ν:

- $[\![x]\!]_\nu = \nu(x)$
- $[\![z]\!]_\nu = z$
- $[\![t_1 + t_2]\!]_\nu = [\![t_1]\!]_\nu + [\![t_2]\!]_\nu$
- $[\![t_1 - t_2]\!]_\nu = [\![t_1]\!]_\nu - [\![t_2]\!]_\nu$
- $[\![t_1 \times t_2]\!]_\nu = [\![t_1]\!]_\nu \times [\![t_2]\!]_\nu$
- $[\![t_1/t_2]\!]_\nu = [\![t_1]\!]_\nu / [\![t_2]\!]_\nu$, and is undefined if $[\![t_2]\!]_\nu == 0$.
- $[\![t_1\%t_2]\!]_\nu = [\![t_1]\!]_\nu \mod [\![t_2]\!]_\nu$, and is undefined if $[\![t_2]\!]_\nu == 0$.

We consider the set of usual *comparison operators* over $\mathbb{Z}$, Comp $= \{<, >, ==, \leq, \geq, \neq\}$. A *guard* is a comparison between two arithmetic expressions, *i.e.*, t_1 op t_2, with $t_1, t_2 \in$ Arith and op $\in$ Comp. We denote Guard the set of these guards. In the formalism presented earlier, Guard will be the set of predicate we allow in FO. The semantic of a guard is the natural one, so we do not detail it here.

We now define the set of instructions we allow in our small language.

```
skip;
x = t;
P1; P2;
if(guard) P1; else P2;
while(guard) P;
assert(guard);
```

In the above, t is in Arith, and *guard* is in Guard. The instructions are the classic C instructions. Assert instruction is a special instruction that if the guard is violated goes to an error state. As we are only concerned with reachability of an error state here, we do not model return statements or

outputs of the program. The only thing we will investigate in our program is that asserts are never violated.

But as simple as this language is, it is still not practical enough to handle in our tool. Rather than giving its semantics, we will therefore explain how to convert a program in an automata-like model (which manipulates memory) to simplify its manipulation. We will thus give the semantic of that automata-like model.

3.2 Control-Flow Automata

A *control-flow automaton* (CFA) is a tuple $(Q, q_i, q_{bad}, \Delta)$ where Q is a finite set of states, $q_i \in Q$ is the initial state, q_{bad} is the final states (representing the "bug" position), and $\Delta \subseteq Q \times \mathrm{Op} \times Q$ is the set of transitions, where Op is the set of possible operations: $\mathrm{Op} = \{\mathrm{skip}\} \cup \{x := t \mid x \in \mathrm{Var}, t \in \mathrm{Arith}\} \cup \mathrm{Guard}$.

Said otherwise, it is an automaton that can manipulate variables with value in $\mathbb{Z}$, and test their values. We will first describe how to represent programs as CFA, and then give their semantics.

3.2.1 Representing a program as a CFA

We will define inductively a translation from programs to CFA. Informally, states of the CFA represents position of the program. Affectations and skip will be represented directly as a single transition. Sequences will be obtained by concatenating the representation of its components. Tests will be represented as two transitions outputing from a single state. While instructions will be represented as a cycle in the automaton. Finally, assert will be represented as tests whose negative branch goes to the bad test, and positive branch continues to the next instruction.

The representations are given in the Figures 3.1, 3.2, 3.3, 3.4 and 3.5. $\neg$ `guard` is obtained by replacing the comparator operation by its opposite ($<$ is opposite of $\geq$, $>$ is the opposite of $\leq$ and $\neq$ is the opposite of $==$). For the sake of simplicity in the induction presentation, we put a lot of skip edges. However, in the concrete automata we will produce, we will collapse these edges whenever it is possible (actually, in a CFA produced by a concrete program, it is always possible to get rid of them). However, we keep skip in the automata for a reason. First, we may give example directly as automaton where skip instruction might help giving a more concise presentation of an algorithm. Secondly, skip may be used to represent non-determinism in programs, whereas it comes from a test that is to complex to formalise in

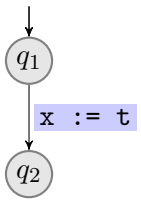

Figure 3.1: The CFA representing `x := t`

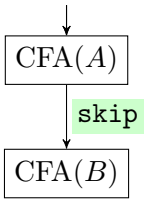

Figure 3.2: The CFA representing `A; B;`

the setting studied, or that we want to model user input (which we cannot make any assumption on).

In Figure 3.6, we give a short program and its representation as a CFA, where skip edges have been collapsed.

3.2.2 Semantic of CFA

We now move to the definition of the semantics of a CFA. That will be done through the notion of configurations, transition relation, executions, and runs.

A *configuration* of a control-flow automaton $\mathcal{A}$ is a pair (q, ν), where q is a state, and ν a valuation in $\mathbb{Z}^{\text{Var}}$.

The *semantic* of an operation op is the relation $[\![\text{op}]\!]$ over pairs of valuations, containing all valuation (ν_1, ν_2) such that ν_2 can be obtained by applying op to ν_1. Formally,

- $[\![\texttt{skip}]\!] = \text{id}$ (the identity relation),
- $[\![\text{guard}]\!] = \{(\nu, \nu) \mid \nu \models \text{guard}\}$,
- $[\![\texttt{x} := \texttt{exp}]\!] = \{(\nu, \nu[x \leftarrow [\![exp]\!]_\nu]) \mid [\![exp]\!]_\nu \text{ is defined}\}$.

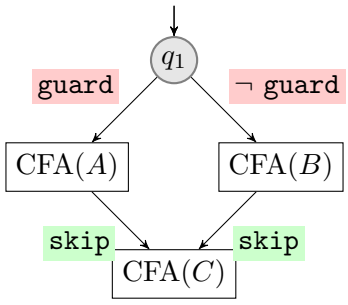

Figure 3.3: The CFA representing `if(guard) then A else B; C;`

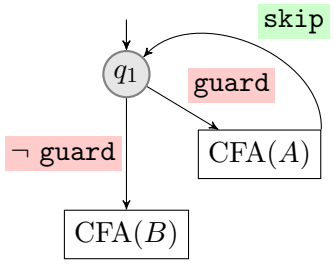

Figure 3.4: The CFA representing `while(guard) A; B;`

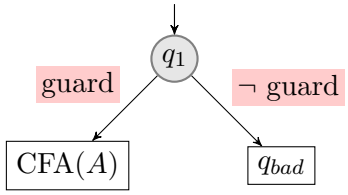

Figure 3.5: The CFA representing `assert(guard); A;`

```
x = 1;
if (y <= 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
assert(x != 0);
```

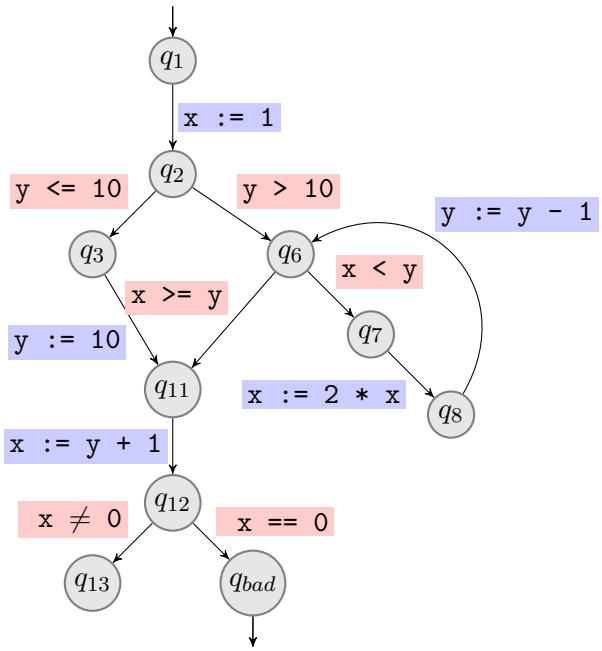

Figure 3.6: A CFA representing a short program.

The semantic of a transition $t = (q, \mathrm{op}, q')$ is the binary relation $\xrightarrow{t}$ over configurations defined as

$$c \xrightarrow{t} c' \Leftrightarrow c = (q, \nu), c' = (q', \nu'), (\nu, \nu') \in [\![\mathrm{op}]\!]$$

The *step relation* $\rightarrow_{\mathcal{A}}$ is the union of all the transition semantics of $\mathcal{A}$: $\bigcup_{t \in \Delta_{\mathcal{A}}} \xrightarrow{t}$.

An *execution* is a sequence $c_0, t_1, c_1, \ldots, t_n, c_n$ alterning configurations c_i and transitions t_i such that $c_{i-1} \xrightarrow{t_i} c_i$ for all $0 < i \leq n$. Such an execution is also written $c_0 \xrightarrow{t_1} c_1 \cdots \xrightarrow{t_n} c_n$ to improve readability, and n is ets *length*.

A path $q_0, \mathrm{op}_1, q_1, \ldots, \mathrm{op}_n, q_n$ is said *executable* if there exist valuations $\rho_0, \ldots, \rho_n \in \mathbb{Z}^{\mathrm{Var}}$ such that $(q_0, \rho_0) \xrightarrow{t_1} (q_1, \rho_1) \cdots \xrightarrow{t_n} (q_n, \rho_n)$ is an execution, with $t_i = (q_{i-1}, \mathrm{op}_i, q_i)$.

Finally, a *run* of a CFA is an execution whose first configuration is initial, i.e., contains the initial state. We do not suppose that there are any restriction in the valuation in the initial configuration in our setting. A run whose last configuration contains q_{bad} is called a *faulty run*.

We can now formulate the variant of the reachability problem we will be studying on CFA:

Input: A CFA $\mathcal{A} = (Q, q_i, q_{bad}, \Delta)$.

Output: Does there exist a faulty run?

Observe that, as already promised, as CFA are actually Turing-complete, this problem is undecidable.

3.3 Encoding the semantics in FO

The next step we need to do to be able to use SMT-solvers to solve the aforementioned problem, is how to encode the CFA semantic in FO.

We first define the formula associated with an instruction. Such a formula will rely two copies of the variables we call here Var_1 and Var_2, and we will consider elements of Var_i to have their name indexed by i. In later section, we might have more copies of Var, but the formulæ defined here will remain valid (up to renaming which we will let implicite). If a formula is indexed by i, we mean that all variables in it are indexed by i.

The simplest formula to define is the one encoding skip, as it does not modify any variable, nor introduce any restriction. Therefore, it is simply:

$$\varphi_{\mathrm{skip}} = \bigwedge_{x \in \mathrm{Var}} x_1 == x_2$$

The formula associated with a guard $\exp \diamond \exp'$ does not modify any variable, but introduces restriction (namely $\exp_1 \diamond \exp'_1$, which can directly be seen as a FO formula), so we add it to the formula :

$$\varphi_{\exp\diamond\exp'} = \bigwedge_{x\in\mathrm{Var}} x_1 == x_2 \wedge \exp_1 \diamond \exp'_1 \wedge \mathrm{side}_{\exp_1} \wedge \mathrm{side}_{\exp'_1}$$

The two side formulæ are here because, as mentionned earlier, Z3 does not handle division by zero, and it must be ensured in the formula no denominator is equal to zero. $\mathrm{side}_{\exp}$ is a conjunction of formula expressing that. We let the detail in exercise, but give an example.

If $\exp = x/(y+3/z)+z/(3-x)$, $\mathrm{side}_{\exp} = (y+3/z \neq 0)\wedge(z \neq 0)\wedge(3-x \neq 0)$.

Finally, the formula associated with an affectation $x := \exp$ is the following:

$$\varphi_{x:=\exp} = \bigwedge_{y\in\mathrm{Var}\setminus\{x\}} y_1 == y_2 \wedge x_2 == \exp_1 \wedge \mathrm{side}_{\exp_1}$$

To encode the semantics of a CFA, we can add variables to represent states. We will consider that those variables are named Q_i (same indices as the variables), which represent that at step i, the variable is Q. The formula representing a transition $t = q \xrightarrow{\mathrm{op}} q'$ is therefore simply:

$$\varphi_t = Q_1 == q \wedge Q_2 == q' \wedge \varphi_{\mathrm{op}}$$

The step formula of the automaton is

$$\varphi_{\mathrm{step}(\mathcal{A})} = \bigvee_{t\in\Delta} \varphi_t$$

We can finally express that a path $t_1, \cdots, t_k$ labels a faulty run with the following formula:

$$Q_0 == q_i \wedge Q_k == q_{bad} \wedge \bigwedge_{0\leq i<k} \phi_{t_i}[i, i+1]$$

where $\phi[i, i+1]$ means the formula ϕ where x_1 is replaced with x_i and x_2 with x_{i+1}. And similarly, the existence of a faulty run of size k is:

$$Q_0 == q_i \wedge Q_k == q_{bad} \wedge \bigwedge_{0\leq i<k} \phi_{\mathrm{step}(\mathcal{A})}[i, i+1]$$

3.3.1 Backward Semantics

As we will see in the next chapter, it might be sometimes better to explore a CFA backwards from the final state, and thus, we also need to encode the backward semantic of a CFA. Fortunately, as the semantics consists in having a different copy of each variable for every step and formulæ relying them, it won't require much work as the formulæ are time-independant. To do so, in what we described earlier, we only need to inverse the role of x_1 and x_2 for each variable. As you may notice, this only modifies the formula encoding the assign statement, as the other are entirely symmetrical.

3.4 Implementation

The practical session linked with this chapter consists in implementing the semantic of operations defined at the begining of Section 3.2.2. It will consist in encoding the semantic of an operation as an FO-formula linking the values of variables in the first valuation and the second valuation. It will rely on translating an arithmetical term into a formula, on which the only real difficulty is that this formula has to contain a part forbidding any denominator the term contain to be equal to zero (as, for technical reason, Z3 ignores division by zero). More details are given in the code to complete and the attached documentation.

Chapter 4

Bounded Model-Checking

This chapter is - finally - devoted to the presentation of the verification technique that is the object of this part of the course. It first present the problem of Bounded Model Checking (BMC), and then two different algorithms to solve it.

This technique has been introduced in the end of the 1990's on finite state systems as a complete algorithm [2] (this reference is not the first one but is a first summary on the technique), and has been quite rapidly extended to infinite state systems (at the obvious cost of becoming incomplete) [3].

In this course, we will present the BMC for infinite state systems as CFA are such systems.

4.1 Principle

4.1.1 The problem and its encoding in FO

Bounded Model Checking is an incomplete method as presented in the first chapter. Informally, it simply consist in checking all runs of a CFA of length at most a given constant k. Given that the number of such runs is finite, the problem is obviously decidable. However, if no run of length at most k is faulty, that is not a guarantee there is no faulty run. That would seem to be contradictory with the soundness requirement we want for our techniques. However, our algorithms will be able to distinguish between the case where no run of length at least k exist (in which case, the algorithm can assure there is no faulty run at all), and the case when at least one run of length at least k exists (in which case, if there is no faulty run of length at most k, we can only say that we don't know if there is a faulty run or not).

Formally, then, the bounded model-checking problem is the following:

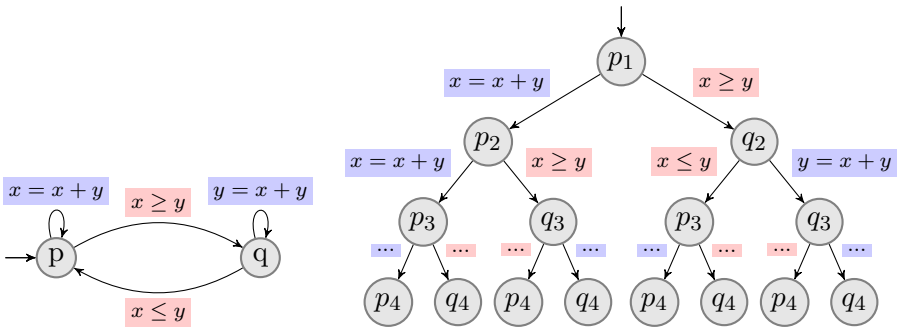

Figure 4.1: A simple CFA and its unfolding up to depth 4

Input: A CFA $\mathcal{A}$, and a natural $k \in \mathbb{N}$.

Output: Does there exist a faulty run of length at most k? If not, does there exist a run of length at least k?

A way of seeing the problem, is to unfold the runs of the CFA up to depth k to obtain a tree (of depth k), and for each node, test if the path represented by it is an actual execution or not. Figure 4.1 presents a system and its unfolding. Of course, the size of such a tree is exponential in k, so if we were to actually build the tree, we would yield a EXPSpace procedure. However, the algorithms we will use do not construct the tree, and can actually limit the complexity to PSpace.

To encode it in with FO-formulae, we consider that there is a copy of Var for each depth of the tree, and the edges of the tree are labelled with formulæ encoding the semantics of the instruction of the transition it represents (as described in Chapter 3). The formula of an edge starting at depth i will relate variables from Var_i and Var_{i+1}. Thus a path will be a run if and only if the formula labelling it is satisfiable.

4.1.2 Backward VS Forward

The more natural way to apply this approach is to start from the initial state and unfold the system forward to check all runs of depth at most k to see if one reaches q_{bad}

There is however another possibility to test that which consists in starting from q_{bad}, and applying transition backward to check if there is an execution ending in q_{bad} of length at most k which is actually a run. As discussed

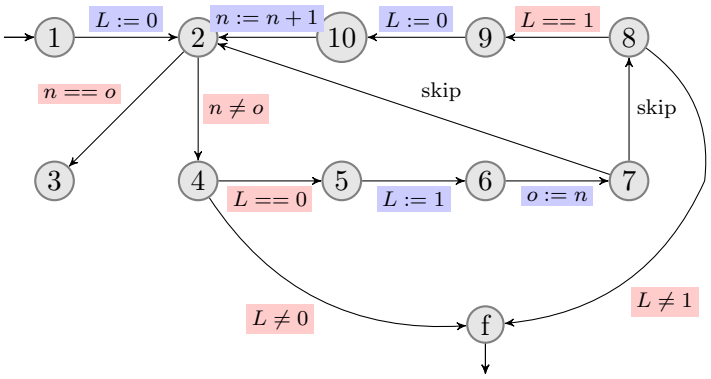

Figure 4.2: A classical lock process

in the previous chapter, in term of translation to a formula, this approach need no more work than the previous one to encode the process in FO.

Both approaches will find a counter-example if there is one, so one might ask: "why bother?". The answer is that in the case no counter-example is found, the response of the algorithms might differ. Indeed, it might be the case that a program is correct and has infinite runs, but that actually no execution of length at least 35 ends in q_{bad} from any configuration. In that case, the forward approach will return a non-exhaustive search, while the backward approach will be able to deem the program correct. Of course, the symmetrical situation being possible, it is worth having the two approach.

In summary, running the approach backward and forward will allow to answer the question on more programs (obviously, not on all of them).

4.1.3 Example: classical lock process

The CFA presented if Figure 4.2 represents a classical lock algorithm (or rather an abstraction of it), that counts the number of times it does something in the critical section and stops when no new data arrives. The L variable represents the lock, while the n and o variable are the counter of the loop (new and old), used to determine if a new data was read in the critical section (the fact of doing something being represented by each skip instruction starting in state 7). It is present in spa as `running_example.aut`.

Figure 4.3 presents the forward unfolding of this CFA. Red nodes represent configurations which are not accessible (i.e., the conjunction of constraints on the path from the root to it is unsatisfiable). On the edges are the constraints introduced by the transition represented. We only display

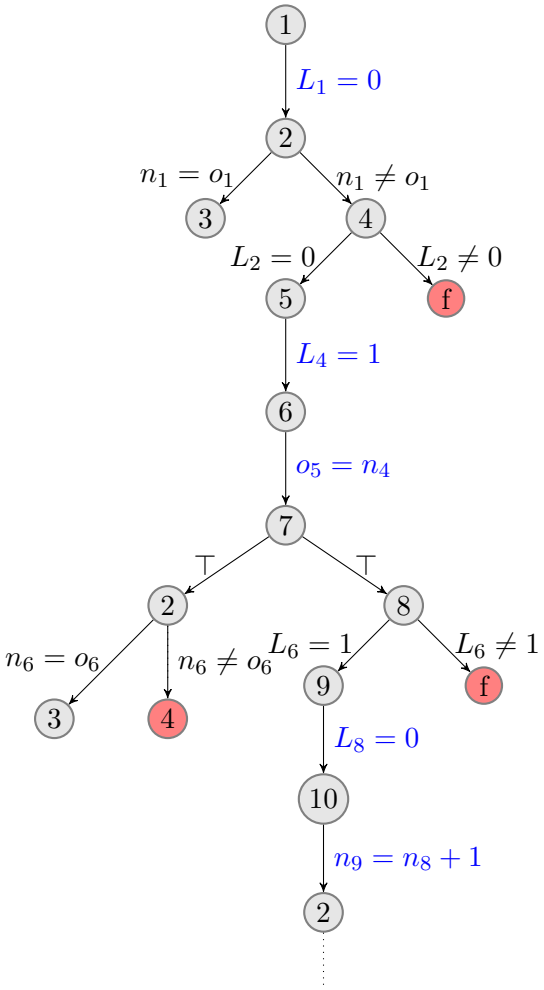

Figure 4.3: The forward unfolding

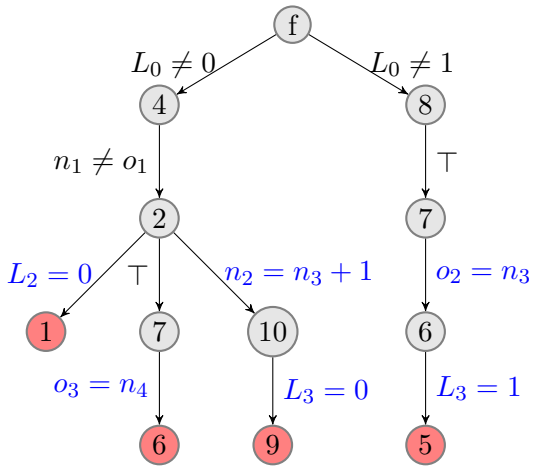

Figure 4.4: The backward unfolding

the part of the formula that is not simply the equality between a variable at a depth and the next. Therefore, if a variable does not appear, it is not modified (i.e., $x_{i+1} = x_i$). The index of a variable is the depth of the node it is associated to (root is at depth 0). Equalities representing affectations are depicted in blue, for clarity.

Similarly, Figure 4.4 represents the backward unfolding of this CFA.

One can observe that it is a case in which the forward BMC will return "unknown" as there are infinite execution of this CFA, while the backward BMC will return correct, as it allows to prove that no execution of length more than 3 can reach q_{bad}.

4.2 Depth First Search Algorithm

4.2.1 The algorithm

The algorithm is presented in Algorithm 2. Given a CFA $\mathcal{A}$ and a bound k, it is called as ForwardDFSBMC($\mathcal{A}, k, \epsilon, q_{in}, 0$). It basically simply performs a DFS on the unfolding described earlier. To do so, it recursively calls itself at each node, for each outputting transition, after having checked that the current node is reachable and not q_{bad}, and that the bound isn't reached (in which cases, the result is sent right away), and compile the result from these calls.

4.2.2 Implementation

You will implement this algorithm, alongside its backward version (which will simply be obtained by reversing the CFA and calling the relevant semantic function).

An immediate optimisation we'll do in the code is to reduce the number of variables. As you have noticed, most variable are unchanged from a node to the next. So, to reduce the number of variables in the SMT-solver, the algorithm will in addition remember for each variable x which depth d it was last modified and use x_d for comparisons. For example, in Figure 4.3, on the edges from node 8, instead of using the variable L_6, we will use L_4 as L is not modified from state 6 to state 8. The managing of these names is already done for you in the code.

Contrary to what was given in the pseudo-code (for clarity), the path will not be an argument of the function you code (for not having useless argument), so the counter-example path will have to be reconstructed while exiting the function with the result "Counter-example found".

4.3 Global Algorithm

To solve this problem, the global search, performed in the forward direction, consists in defining a SMT-formula $\varphi_{\text{step}}(q, X, q', X')$ equivalent to the $\rightarrow_{\mathcal{A}}$ relation, and checking successively for every i between 0 and k whether the following formula[1] is satisfiable:

$$\psi_i(q_0, X_0, \cdots, q_i, X_i) \stackrel{\text{def}}{=} q_0 = q_{in} \wedge \bigwedge_{j=1}^{i} \varphi_{\text{step}}(q_{j-1}, X_{j-1}, q_j, X_j) \wedge q_i = q_{bad}$$

This algorithm is presented in detail in Algorithm 3. The algorithm stops as soon as $\psi_i(q_0, X_0, \ldots, q_i, X_i)$ is satisfiable and, in that case, it returns the feasible path from q_{in} to q_{bad} represented by a model of the formula. The algorithm also stops after i steps if no execution of length i exists starting from q_{in} (and, in that case, it returns that the program contains no execution from q_{in} to q_{bad}).

4.3.1 Implementation

You will implement this algorithm (alongside its backward version which, like for the forward case is simply a detail).

[1] The formula ψ_0 is $q_0 = q_{in} \wedge q_0 = q_{bad}$.

Algorithm 2: `ForwardDFSBMC`$(\mathcal{A}, k, \tau, q, \ell)$.

Input: A program automaton $\mathcal{A}$, a bound $k \in \mathbb{N}$, a path τ, a current state q and a current depth ℓ.

Output: Whether there exists an execution of length at most k from q_{in} to q_{bad}.

```
1  if k > ℓ then
2      return Non-exhaustive
3  if τ is not an execution then
4      return Exhaustive
5  if q = q_bad then
6      return Counter-example(τ)
7  exhaustive ← ⊤
8  foreach (q, op, q') ∈ Δ do
9      acc ← ForwardDFSBMC(A, k, τ(q, op, q'), q', ℓ + 1)
10     if acc == Counter-example(_) then
11         return acc
12     exhaustive ← exhaustive ∧ (acc ==Exhaustive)
13 if exhaustive then
14     return Exhaustive
15 else
16     return Non-exhaustive
```

To reconstruct the counter-example path when there is one, you will need to enrich the formula with more information than what is described here.

Algorithm 3: `ForwardGlobalBMC`$(\mathcal{A}, k)$.

Input: A program automaton $\mathcal{A}$ and a bound $k \in \mathbb{N}$.
Output: Whether there exists an execution of length at most k from q_{in} to q_{bad}.

```
1  i ← 0
2  while i ≤ k do
3      ψ ← q_0 = q_in ∧ ⋀_{j=1}^{i} φ_step(q_{j−1}, X_{j−1}, q_j, X_j)
4      if ψ is not satisfiable then
5          return "no execution from q_in to q_bad"
6      ψ ← ψ ∧ q_i = q_bad
7      if ψ is satisfiable then
8          Extract from a model of ψ a feasible path π from q_in to q_bad
9          return "feasible path π from q_in to q_bad"
10     i ← i + 1
11 return "no execution of length at most k from q_in to q_bad"
```

Bibliography

[1] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

[3] Tobias Schüle and Klaus Schneider. Bounded model checking of infinite state systems. *Formal Methods Syst. Des.*, 30(1):51–81, 2007.